Approximate parallel scheduling. Part II: Applications to optimal parallel graph algorithms in logarithmic time

by

Richard Cole‡
Uzi Vishkin†

Ultracomputer Note #119
Computer Science Technical Report #291
April, 1987

Approximate parallel scheduling. Part II: Applications to optimal parallel graph algorithms in logarithmic time

by

Richard Cole‡
Uzi Vishkin†

Ultracomputer Note #119
Computer Science Technical Report #291
April, 1987

‡ This research was supported in part by NSF grant DCR-84-01633, ONR grant N00014-85-K-0046 and by an IBM faculty development award.

† This research was supported in part by NSF grants NSF-CCR-8615337 and NSF-DCR-8413359, ONR grant N00014-85-K-0046 and by the Applied Mathematical Science subprogram of the office of Energy Research, U.S. Department of Energy under contract number DE-AC02-76ER03077. **Current Address:** Tel Aviv University, Dept. of Computer Science, School of Mathematical Science, Tel Aviv 69 978, Israel.

ABSTRACT

Part I of this paper presented a novel technique for approximate parallel scheduling and a new logarithmic time optimal parallel algorithm for the list ranking problem. In this part, we give a new logarithmic time parallel (PRAM) algorithm for computing the connected components of undirected graphs which uses this scheduling technique. The connectivity algorithm is optimal unless $m = o(n \log^* n)$††, in graphs of n vertices and m edges. Using known results, this new algorithms imply logarithmic time optimal parallel algorithms for a number of other graph problems, including: biconnectivity, Euler tours, strong orientation and *st*-numbering. Another contribution of the present paper is a parallel union/find algorithm.

To the best of our knowledge, this paper presents for the first time optimal, logarithmic time, parallel algorithms for problems on graphs.

1. Introduction

The models of parallel computation used in this paper are all members of the parallel random access machine (PRAM) family. A PRAM employs p synchronous processors all having access to a common memory. An exclusive-read exclusive-write (EREW) PRAM does not allow simultaneous access by more than one processor to the same memory location for read or write purposes. A concurrent-read exclusive-write (CREW) PRAM allows simultaneous access for reads but not writes. A concurrent-read concurrent-write (CRCW) PRAM allows concurrent access for both reads and writes. For the CRCW PRAM, we assume that if several processors attempt to write simultaneously at the same memory location then one of them succeeds but we do not know in advance which one. See [Vi-83] for a survey of results concerning PRAMs.

Let $Seq(n)$ be the fastest known worst-case running time of a sequential algorithm, where n is the length of the input for the problem at hand. Obviously, the best upper bound on the parallel time achievable using p processors, without improving the sequential result, is of the form $O(Seq(n)/p)$. A parallel algorithm that achieves this running time is said to have *optimal speed-up* or more simply to be *optimal*. A primary goal in parallel computation is to design optimal algorithms that also run as fast as possible.

Most of the problems we consider can be solved by parallel algorithms that obey the following framework. Given an input of size n the parallel algorithm employs a *reducing*

††$\log^{(k)}$ denotes the kth iterate of the log function and $\log^*$ denotes the least i such that $\log^{(i)} n \leq 2$.

procedure to produce a smaller instance of the same problem (of size $\leq n/2$, say). The smaller problem is solved recursively until this brings us below some threshold for the size of the problem. An alternative procedure is then used to complete the parallel algorithm. We refer the reader to [CV-86d] where this algorithmic technique, which is called accelerating cascades, is discussed. Typically, we need to reschedule the processors in order to apply the reducing procedure efficiently to the smaller sized problem. Suppose the input for a problem of size n is given in an array of size n. A natural approach is to compress the smaller problem instance into a smaller array, of size $\leq n/2$. This is often done using a standard prefix sum algorithm (it takes $O(\log n)$ time on $n/\log n$ processors to compute the prefix sums for n inputs stored in an array). Thus if we need to reschedule the processors repeatedly it is unclear how to achieve logarithmic time. Sometimes the rescheduling need not be performed very often: [CV-86a,C-86] show that for some problems (list ranking and selection) $\log^* n$ reschedulings suffice. Alternatively, one can use a fast random algorithm to perform the rescheduling, or at least an approximate rescheduling. (By approximate rescheduling we mean that we may not be able to partition the work evenly among the processors, but only approximately evenly.) Thus the need for rescheduling does not preclude $O(\log n)$ time optimal random algorithms. Let us mention the main contributions of Part I of this research, the paper [CV-86c]). Part I provides an algorithm for performing approximate rescheduling deterministically in $O(1)$ time. This is used to solve a novel scheduling problem. The solution to the scheduling problem leads to a logarithmic time optimal deterministic parallel algorithm for list ranking. In the present paper, a related rescheduling procedure will be one of the tools that leads to a logarithmic time connectivity algorithm which is optimal unless the graph is extremely sparse.

We identify the following *duration-unknown* task scheduling problem. n tasks are given, each of length between 1 and $e \log n$, e a constant; the total length of the tasks is bounded by cn, c a constant. (A task can be thought of as a program.) However, we do not know, in advance, the lengths of the individual tasks; in fact, they may vary, depending on the order of execution of the tasks. The problem is to schedule the n tasks on an EREW PRAM of $n/\log n$ processors so that the tasks are completed in $O(\log n)$ time; this problem is solved in Part I.

We now discuss how to design algorithms that take advantage of this task scheduling algorithm. Given a problem, our job is to design a "protocol" for solving the problem by using a set of short tasks (each of length between 1 and $e \log n$). This provides an important new opportunity for the designer of a protocol which is based on using the scheduling algorithm: the designer of the protocol need not know anything about the order of execution of the tasks. Such an opportunity for designing parallel tasks, without knowing in advance their lengths, with the guarantee that they will be scheduled efficiently, sounds very promising. However, this opportunity cannot be separated from a considerable difficulty in designing such a protocol: we have no control over the order of execution of the tasks, so we must ensure that the protocol works correctly regardless of the order of execution. We note that this style of protocol design may be useful for distributed systems that are not tightly synchronized; here too, we have to be sure that the protocol works correctly regardless of the order of execution. Part I demonstrates how to design such a protocol for the list-ranking problem. In Section 3.3, we demonstrate how to design such a protocol for a problem which occurs in our parallel connectivity algorithm.

The main problem considered in this paper is *graph connectivity*, which is defined as follows:

Input: An undirected graph with n vertices and m edges.

The problem: Find the connected components of the graph.

Results: We obtain the following efficient parallel algorithms, **optimal** for $m \geq n \log^* n$:

1. On the CRCW PRAM: $T = O(\log n)$ time using $O((n+m)\alpha(m, n)/T)$ processors, where $\alpha(m, n)$ is the inverse Ackerman function. The algorithm requires space $O(\min[n^2, m\, n^\epsilon])$, where ϵ can be any constant satisfying $0 < \epsilon < 1$.

2. On the CREW PRAM: $T = O(\log^2 n)$ time using $O((n+m)\alpha(m, n)/T)$ processors. The algorithm requires linear ($O(n+m)$) space.

The previous best results for computing connected components are: $O(\log n)$ time using $O(m)$ processors on the CRCW PRAM [SV-82], and $O(\log^2 n)$ time using $n^2/\log^2 n$ processors on the CREW PRAM [CLC-82] or on the CRCW PRAM [Vi-84]. [KRS-85] gave an efficient algorithm for relatively slow times; specifically, they achieved $O\left(\frac{m+n}{p} \cdot \frac{\log (m+n)}{\log (m+n)/p}\right)$ time using $p < (n+m)/2$ processors, and space $O(pn + m)$ on the EREW PRAM. [G-86] recently gave a randomized connectivity algorithm which runs, with high probability, in logarithmic time using an optimal number of processors.

The literature gives quite a few efficient parallel algorithms for undirected graph problems which essentially reduce a graph problem into the problem of finding a (any) spanning forest in a graph. Fortunately, our connectivity algorithms also finds a spanning forest within the same time and processor efficiencies. This leads, without too much effort, to parallel algorithms which run in time $O(\log n)$ using $\frac{n\log n + m}{\log n}$ processors for the following problems:

1. Finding biconnected components of undirected graphs, using the algorithm of [TV-85].

2. Orienting the edges of a connected bridgeless undirected graph, so that the resulting directed graph is strongly connected, using the algorithm of [Vi-85].

3. Ear decomposition and finding *st*-numbering of biconnected graphs, using the algorithm of [MSV-86].

4. Finding Euler tours in directed and undirected graphs (or determining that they do not exist), using the algorithm of [AV-84] and a logarithmic time optimal list ranking algorithm. Such algorithms were given in Part I and [CV-86f].

With some additional effort, which includes applying the new parallel lowest common ancestor algorithm of [ScV-87] and the logarithmic time optimal parallel list ranking of [CV-86f], we extend these logarithmic time optimal speed up results to sparser graphs, where $m = o(n \log n)$. We will not provide the details of these improved algorithms, for it would require us to describe anew parts of these other papers, and we want to keep this presentation within reasonable length. Also the descriptions of these improvements, particularly for the case $m = o(n \log n)$, are quite tedious, and we doubt whether such lengthly descriptions merit publication.

Previous parallel algorithms for these four problems were also given in [At-84], [AIS-84], [Lo-85], [SJ-81] and [TC-84].

We make several contributions here. Recall that Part I provides a new approach to the rescheduling problem. This approach is applied in the connectivity algorithm of the present paper. We provide a completely new approach for the connectivity problem. Previous attempts at parallel algorithms for the connectivity algorithm consisted of applying the connectivity computation procedure directly to the whole input graph. These attempts failed to produce optimal connectivity algorithms for sparse graphs since all known connectivity procedures need more than a linear number of operations for such graphs. However, the new approach circumvents this problem by applying a connectivity computation procedure to relatively small subgraphs at each time. The lack of a logarithmic time optimal parallel connectivity algorithm had been the only obstacle for getting similar results for several graph problems, as mentioned above. We are not familiar with any previous logarithmic time optimal parallel algorithm for any graph problem. Moreover, the only known poly-log time optimal parallel algorithms are based on the assumption that the graph is given by its adjacency matrix, in which case the fastest serial algorithms need $O(n^2)$ time. Such algorithms are based on an $O(\log^2 n)$ time parallel connectivity algorithm, which uses $\frac{n^2}{\log^2 n}$ processors.

Such algorithms were given in [CLC-82] and [Vi-84].

Preliminary versions of this work appeared as parts of [CV-86d] and [CV-86e].

In Section 3 we describe the new CRCW connectivity algorithm. Subsection 3.1 reviews previous work and, thereby, provides motivation for the following subsections. In the beginning of Subsection 3.2, we overview the other subsections. In Section 4 we outline the CREW connectivity algorithm. The Appendix gives a parallel union/find algorithm.

2. Preliminaries

We give below a useful and simple scheme, due to Brent, for designing parallel algorithms.

Theorem (Brent). Any synchronous parallel algorithm of time t that consists of a total of x elementary operations can be implemented by p processors in time $\lceil x/p \rceil + t$.

Remark. Brent's Theorem is stated for parallel models of computation where not all computational overheads are taken into account. Specifically, the implementation problem of assigning the processors to their jobs is ignored. Often, in the present paper, it is straightforward to overcome this implementation problem without increasing the running time or the number of processors in order of magnitude. Therefore, we allow ourselves to switch freely from a result of the form "$O(x)$ operations and $O(t)$ time" to "x/t processors and $O(t)$ time" (and vice versa, which is always correct). However, we avoided doing this where there are difficulties with these implementation problems.

3. Graph Connectivity

3.1. Basic Techniques and Previous Work

We start with a few definitions. Essentially, there exist two parallel poly-log time connectivity algorithms. [HCS-79] yields $O(\log^2|V|)$ time and [SV-82]) yields $O(\log|V|)$ time. (We view [CLC-82], [Vi-84] and [W-79] as implementations of the HCS algorithm and [AS-83] as an implementation of the SV algorithm). We review these algorithms briefly and discuss the obstacles to deriving optimal speed-up implementations from them.

Our problem is to compute the connected components of a graph $G=(V,E)$ which is given as follows. *Input form*. Let $V = \{1,\ldots,n\}$ and $|E|=m$. We assume that the edges are given in a vector of length $2m$. The vector contains first all the edges incident on vertex 1, then all the edges incident on vertex 2, and so on. Each edge appears twice in this vector.

Definitions.

(1) A *rooted tree* is a directed graph satisfying:

(a) The undirected graph which is obtained by removing directions from the edges is a tree.

(b) It has a vertex r called the *root* such that there exists a directed path from each vertex to r.

(2) A *rooted star* is a rooted tree in which the path from each vertex to the root comprises (at most) one edge.

The following is common both to the HCS and SV connectivity algorithms and to the new connectivity algorithm presented here. At each step during the algorithms each vertex v has a pointer field D through which it points to another vertex or to no vertex. One can regard the directed edge $(v, D(v))$ as a directed edge in an auxiliary graph, called the *pointer graph*. Initially, for each vertex v, $D(v)$ points to no vertex. The pointer graph keeps changing during the course of the algorithms. However, at each step of each of these three algorithms the pointer graph consists of rooted trees. It will be convenient to refer to a set of vertices comprising a tree as a *supervertex*. Sometimes, we identify a supervertex with the root of its tree. No confusion will arise. As the algorithms proceed, the number of trees (supervertices) decreases. This is achieved by (possibly simultaneous) *hooking* operations. In each hooking a root r of a tree is 'hooked' onto a vertex v of another tree (that is, $D(r) := v$). A careful look at each of these connectivity algorithms reveals that:

1. Each such hooking is performed only after the algorithm "identified" an edge connecting a vertex in the supervertex of r with a vertex in the supervertex of v. Let us call such connecting edge the *causing edge* of its hooking. We illustrate this notion of causing edge in the description of the HCS algorithm given below.

2. (*The spanning forest property*) For each supervertex, consider the collection of the causing edges that connect pairs of its vertices. This collection forms a spanning tree of the supervertex with respect to its vertices. Specifically, the collection of the causing edges, throughout each of these connectivity algorithms, forms a spanning forest of the input graph.

The trees are also subject to a *shortcut* operation. That is, for every vertex v of the tree,

```
if $D(D(v))$ is some vertex (as opposed to no vertex)
  then $D(v) := D(D(v))$.
```

The shortcut operation (approximately) halves the height of a tree. Shortcuts do not introduce cycles into the pointer graph, as can be readily verified. Simultaneous hookings are performed in each of these algorithms in such a way that no cycles are introduced into the pointer graph. The algorithms also use the following graph. Each edge (u,v) in the input graph induces an edge connecting the supervertex containing u with the supervertex containing v. The graph whose vertices are the supervertices and whose edges are these induced edges is called the *supervertex graph*.

At the end of each of these algorithms the vertices of each connected component form a rooted star (which is, in particular, a single supervertex) in the pointer graph. As a result, a single processor can answer a query of the form "do vertices v and w belong to the same connected component?" in constant time.

The HCS parallel connectivity algorithm works in $O(\log n)$ iterations. Upon starting an iteration each supervertex is represented by a rooted star in the pointer graph. Each root hooks itself onto a minimal root which is adjacent to it in the supervertex graph. In case two roots are hooked on one another, we cancel the hooking of the smaller (numbered) root. As a result several rooted stars form a rooted tree; the root of one of these rooted stars becomes the root of the new tree. An iteration finishes with $O(\log n)$ shortcuts. It remains to identify the causing edges in this algorithm. In the HCS algorithm a root-hooks itself onto a minimal adjacent root. There might be more than one edge connecting the two supervertices. One of them, the one that actually induced the hooking in the course of the computation, is the causing edge of the hooking.

The SV parallel algorithm also works in $O(\log n)$ iterations. Unlike the HCS algorithm: (i) An iteration of SV takes constant time, and (ii) The pointer graph at the beginning of an iteration is a collection of rooted trees (which are not necessarily stars). In principle, an iteration consists of the following steps.

(1) Each rooted star is hooked onto a smaller vertex that is adjacent to some vertex of its supervertex (if there is any such smaller vertex).

(2) Consider the rooted stars, which did not hook and were not hooked upon in step (1); each such rooted star is hooked onto a vertex that is adjacent to some vertex of its supervertex.

(3) Shortcuts.

The algorithm employs a processor for each vertex and each edge of the graph. This amounts to $n+m$ processors. [SV-82] shows that the total height of "still active" trees decreases by a factor of at least 1/3 per iteration, implying that only $O(\log n)$ iterations are needed and, therefore, the algorithm runs in $O(\log n)$ time. The algorithm does not achieve optimal speed up.

3.2. The New Algorithm

Our new algorithm for finding connected components runs in time $O(\log n)$. It achieves optimal speed up for graphs with $m \geq n \log^* n$, and almost optimal speed up in general. The contribution here is as follows. We completely redesigned the SV parallel connectivity algorithm to exploit the new scheduling procedure of Part I. We also take advantage of a new parallel prefix sum algorithm, given in [CV-87], and a new parallel union/find algorithm (given in the Appendix).

The connectivity algorithm we give has two parts.

The second (and more basic) part is an algorithm for relatively dense graphs ($m \geq n \log n \log^{(3)} n$); we call this the *main* connectivity algorithm. The main connectivity algorithm can be viewed as a two level improvement in efficiency relative to the SV algorithm. The first level achieves optimal speed up with a parallel time of $O(\log n \log^{(3)} n)$; the second level achieves optimal speed up with a time of $O(\log n)$. The second level is asymptotically better; however, it does involve the use of expander graphs and the "big oh" notation hides considerably larger constants. The first level is described later in this Section and the second level is described in Section 3.3.

The first part is a reduction procedure which (essentially) reduces the general case to the dense case. The reduction procedure requires $O((m + n)\alpha(m,n))$ operations and $O(\log n)$ time. It is described in Section 3.3. We remark that for $m \geq n \log^* n$, for example, it is optimal. Incidentally, the reduction procedure does not involve expander graphs or similar constructs. The presence of the $\alpha(m,n)$ term is due to the use of a parallel union/find algorithm, which is of interest in its own right.

High-level description of the main connectivity algorithm.

We mention the opportunity and main difficulty in deriving optimal speed up from the SV algorithm. In the SV algorithm, a processor is standing by each edge. For each processor, there is at most one step during the whole algorithm during which its edge is used for hooking. However, the difficulty is that we do not know in advance when this step comes.

The main connectivity algorithm applies the Shiloach/Vishkin connectivity algorithm. Informally, the $O(\log n)$ iterations of this algorithm are now divided into $O(\log\log n)$ phases. Roughly speaking, each phase consists of about as many iterations as all the preceding phases put together. Let us elaborate on this. The specific design of the phases will be clearer if we keep in mind the amortized complexity analysis of the algorithm. The complexity analysis will upper bound the time and number of operations of the whole algorithm.

The analysis uses four kinds of "budgets" for time and number of operations:
(1) An *increasing* budget (for each phase). The increasing budget for each of the two items (namely, time and number of operations) increases by a multiplicative factor from phase to phase.
(2) A *fixed* budget (for each phase). The fixed budget will be the same for each phase.
(3) A *general-pool* budget (for the whole algorithm).
(4) *Miscellaneous* budgets (for each phase). These are described as they are needed.
The *total* budget for the whole algorithm is the sum of these budgets. The partition into phases enables two major changes with respect to the Shiloach/Vishkin algorithm. Below we outline these changes, and how they are paid for by the various budgets.

First, in order to reduce the total number of operations (and, thereby, obtain optimality) each phase applies the Shiloach/Vishkin algorithm only to subsets of the edges. (This requires an edge selection procedure. Part of the budget for the edge selection of each phase comes from the general-pool budget which is independent of the budget for the phase. In other words, the edge selection procedure has an interesting amortized complexity analysis.)

Second, at the end of each phase, the Shiloach/Vishkin algorithm is interrupted in order to contract some of the "small" rooted trees it has constructed to stars. Specifically, the *size* of a supervertex is the number of edges (and not vertices) incident on its vertices; the next phase "expects" that all supervertices whose size is at most some threshold, which depends only on the phase number, to be stars. This contraction is not applied to rooted trees representing supervertices which are larger than this threshold (to be called "large" supervertices). The rationale being that:
1. As with Shiloach/Vishkin we would like our algorithm to take $O(\log n)$ time. For this we need to keep increasing the size of the supervertices at each phase. However, large enough supervertices may remain unchanged until a later phase. Therefore, no damage will occur by excluding them from contraction at this phase.
2. The contraction is expensive in terms of time. (Part of this contraction includes combining together the adjacency lists of the vertices that comprise the same supervertex). The increasing budget for time and number of operations in the present phase may not be big enough for the contraction of a large supervertex. However, later phases have larger increasing budgets, so let them do the contraction.

We have not yet mentioned the role of the fixed budget. Each phase ends up with a clean-up step which prepares the data structures of the algorithm for the next phase. The fixed budget pays for this clean-up step.

To simplify the exposition we assume that the graph comprises a single connected component. In addition, for the purposes of the analysis, it is useful to guarantee that each vertex has degree at least $\log n \log^{(3)} n$. To achieve this, we add up to $\log n \log^{(3)} n/2$ self loops per vertex. We add at most $m/2$ edges (recall that, by assumption, $m \geq n \log n \log^{(3)} n$).

The algorithm proceeds in phases. Each phase is characterized by the following input/output relation.

Input: Each supervertex has $\geq d^2$ incident edges from G, where d is some positive integer, and an edge with both endpoints in the supervertex is counted twice.
Output: Each supervertex has $\geq d^3$ incident edges from G.
Observation: There are at most $3m$ incident edges in the graph (where each edge is counted twice, once for each endpoint).

Initially, $d = (\log n \log^{(3)} n)^{1/2}$. After a further $O(\log \log n)$ phases there is only one supervertex which comprises all the vertices in the graph.

Before describing a phase, we need a few definitions for classifying the edges of G with respect to the supervertex graph. Given a supervertex graph, an edge of G is *redundant* (with respect to the supervertex graph) if both its endpoints lie in the same supervertex. An edge is an *outedge* if it is not redundant. If several outedges connect the same pair of supervertices we arbitrarily choose one of these outedges to be the *actual* outedge; the other outedges are called *duplicate* outedges. The *degree* of a supervertex v is defined to be the number of its incident actual outedges. Also, in each phase we classify the supervertices as *large* or *growing*; large supervertices are known to have at least d^3 incident edges. A phase proceeds as follows.

Step 1. Select an edge set such that

(i) For each growing supervertex of degree $\leq d$, all the actual outedges are selected.

(ii) For each growing supervertex of degree $\geq d$, exactly d actual outedges are selected.

Step 2. Run the Shiloach/Vishkin connectivity algorithm for $O(\log d)$ steps on the graph induced by the edges selected in Step 1.

Step 3. Contract some of the rooted trees constructed in step 2 to stars. Any new supervertex (a rooted tree from Step 2) with at most $d^{4.5}$ incident edges will be contracted to a star.

Contracting all supervertices with at most $d^{4.5}$ incident edges guarantees that any supervertex that is not contracted to a star in step 3 is *large* for the next phase. If a supervertex has sufficiently many incident edges it may be large for several consecutive phases.

The Shiloach/Vishkin algorithm needs to be modified for use in step 2 to account for the presence of large supervertices. We modify it as follows: any supervertex that hooks itself to a large supervertex itself becomes large and does not participate further in step 2.

Observations

a) In Step 1, $O(m/d)$ edges are selected. (Since the input to the phase has $\leq 3m/d^2$ supervertices).

b) Consider the components of the graph comprising the supervertices and the edge set selected in Step 1, above. Each supervertex of degree less than d has all its incident actual outedges selected. Thus, either each component includes a supervertex with at least d actual outedges, or it includes a large supervertex, or there is just one component. As we show (in the description of Step 2) each supervertex formed in Step 2 either comprises $\geq d$ input supervertices (and so has $\geq d^3$ incident edges), or it contains a large input supervertex (and so has $\geq d^3$ incident edges), or it comprises all the vertices.

The procedure for reducing the general connectivity problem to the case $m \geq n \log n \log^{(3)}n$, given in Section 3.3, has the same structure as the main connectivity algorithm. However, steps 1 and 3 are implemented somewhat differently.

As we mentioned above, the new contributions here are:

1. A complete resetting of the SV algorithm which enables it to be applied only to small enough subgraphs and, thereby, reduces the total number of operations performed, so that the algorithm is optimal except when m is $o(n\log^* n)$. An earlier version of the main parallel connectivity algorithm, which required the same number of operations as here, was given in [CV-86a]. However, the time upper bound achieved there was $O(\log n \log^{(2)}n \log^{(3)}n)$.

2. To exploit two scheduling results to save a factor of $\log^{(2)}n \log^{(3)}n$ and, thereby, derive an $O(\log n)$ time upper bound for the connectivity algorithm.

2.1. The first scheduling result is a consequence of a procedure for computing the prefix sums of n numbers, each of $\log n$ bits (the procedure is an optimal CRCW PRAM algorithm that performs $O(n)$ operations in $O(\log n/\log^{(2)}n)$ time). The processor allocation (scheduling) for the connectivity algorithm repeatedly uses this prefix sum procedure. This save a factor of $\log^{(2)}n$ in the time upper bound, with respect to the standard optimal prefix sum parallel algorithm which runs in $O(\log n)$ time. This new prefix sum procedure is described in [CV-87].

2.2. The second scheduling result is the procedure of Part 1 for performing approximate redistribution of objects. This saves a factor of $\log^{(3)}n$. The object redistribution procedure is only needed for step 1, the edge selection.

All these savings together lead to the $O(\log n)$ time optimal connectivity algorithm.

We describe in detail and analyze each step in turn.

The edge selection (Step 1)

Here, we give a procedure for edge selection that performs $O(m)$ operations in time $O(\log n \log^{(3)} n)$ over the course of the whole algorithm. In Section 3.3 we show how to reduce this to $O(\log n)$ time, without changing this operation count.

At the start of the algorithm, the edges incident on each vertex are divided into clusters of $\log n \log^{(3)} n$ edges each (this is justified by the addition of up to $\log n \log^{(3)} n/2$ self loops to each vertex at the start of the algorithm). During the course of the algorithm edges are deleted from the clusters; no edges are added to any cluster. Thus, in general, the edges incident on each supervertex are divided into *clusters* of at most $\log n \log^{(3)} n$ edges each. There are at most $\dfrac{3m}{\log n \log^{(3)} n}$ clusters.

Each cluster is divided into *blocks* of $\log n/\log^{(2)} n$ edges. There is at most one incomplete block (i.e. having insufficiently many edges) per cluster. The division into blocks may change from phase to phase.

Remark: Because of the addition of self loops it is necessary to permit one cluster per vertex to contain one additional edge. Thus, initially, we allow each cluster to contain one block with $\log n/\log^{(2)} n + 1$ edges.

Data structures. $T(v)$ is the name of the supervertex currently containing vertex v of G (T is the *vertex table*). $ACTUAL(u, v)$ is an $n \times n$ array (in fact, as we explain at the end of this section, ACTUAL can be implemented in space $O(\min[n^2, m\, n^\epsilon])$ for any fixed $\epsilon > 0$). It records if an edge connecting supervertex u and supervertex v has been found in the current phase. Specifically, $ACTUAL(u,v)$ comprises a pointer to an edge plus a timestamp; the timestamp is a phase number. The cluster headers (i.e. names) are stored in a single array, with the clusters for each growing supervertex occupying a contiguous portion of the array. For each cluster, we store the headers of its blocks in an array of size $\log^{(2)} n \log^{(3)} n$. For each block we store its edges in an array of size $\log n/\log^{(2)} n$. Finally, for each supervertex (growing or large), we keep its cluster headers in a circular list.

Let C be a cluster belonging to growing supervertex v. We say C is *active* for the current iteration of Step 1.1 (below) of the edge selection procedure unless either we have found d actual outedges incident on v, or we have checked all the edges in C. v is active if any of its clusters are active. Let c be the number of clusters at hand, and let c_A (resp. c'_A)

be the number of active clusters at the start (resp. end) of the current iteration of Step 1.1.

The procedure below assumes $p = \dfrac{3m}{\log n \log^{(3)} n}$ processors are available.

Step 1.1. while $c_A \geq 0$ **do**

1.1.1 For each active cluster, process $\lfloor p/c_A \rfloor$ blocks.

Notice that determining the number of active clusters requires a summation algorithm and hence uses time $O(\log n/\log^{(2)} n)$, while assigning the processors requires a prefix sum algorithm which takes the same time.

Processing a block using a single processor: For each edge $e=(i, j)$ in the block, we determine whether it is redundant, duplicate, or actual. In the first two cases we eliminate the edge from further consideration; in the last case we add the edge to the set of selected edges. We proceed as follows.

a) Let $u=T(i)$, $v=T(j)$. If $u = v$, the edge is redundant. If not, continue with step b.

b) Check $ACTUAL(u, v)$. If an edge is recorded with the current timestamp then e is a duplicate outedge. Otherwise continue with step c.

c) Edge e attempts to write its address and the current phase number to $ACTUAL(u, v)$. If the write is successful e is selected (e is an actual outedge); otherwise e is a duplicate outedge.

1.1.2 For each supervertex, determine if it is still active. (This step is easily performed using a parallel summation algorithm, with respect to the cluster headers.)

end

Step 1.2. For each growing supervertex, among the edges selected in Step 1.1 select d (or all, if $< d$ are available). The selected edges are placed in an array of size $O(m/d)$ (this implies we assign serial numbers to the edges, which is done using a parallel prefix sum computation with respect to the cluster headers).

Analysis. We note that each iteration of Step 1.1 takes time $O(\log n/\log^{(2)} n)$. Also, we observe that in the first iteration (of each phase), at least one block per cluster is processed (for $c \leq \dfrac{3m}{\log n \log^{(3)} n}$ and so $p/c \geq 1$). We show that over the whole algorithm, these iterations perform $O(m)$ operations in time $O(\log n \log^{(3)} n)$. To see this, we divide the iterations into three types: *reducing iterations*, *eliminating iterations*, and *selecting iterations*.

An iteration is *reducing* if $c'_A \leq 1/2 c_A$. Below, we show that, per phase, there are at most $O(\log^{(3)} n)$ reducing iterations (Claim 1). Thus, per phase, the reducing iterations use

$O(\log n \log^{(3)}n/\log^{(2)}n)$ time, and $O(m/\log^{(2)}n)$ operations (to obtain this multiply p processors by $O(\log^{(3)}n)$ iterations by $O(\log n/\log^{(2)}n)$ time per iteration). This is charged to the fixed budget. Over the whole algorithm this is $O(\log n \log^{(3)}n)$ time and $O(m)$ operations.

An iteration is *eliminating* if two properties hold. First, $c'_A > 1/2\, c_A$ (i.e. it is not reducing). Second, consider the clusters that remain active at the end of the iteration; consider the edges in these clusters that were processed during the iteration. If at least half of these edges were eliminated then the iteration is eliminating. As we show below, the number of edges eliminated in an eliminating iteration is at least $3m/(4\log^{(2)}n \cdot \log^{(3)}n)$ (Claim 2). But at most $3m$ edges can be eliminated (each edge in both directions). Thus there are at most $4 \log^{(2)}n \log^{(3)}n$ eliminating iterations over the course of the algorithm. The cost of these iterations ($O(\log n \log^{(3)}n)$ time and $O(m)$ operations) is charged to the general pool budget.

An iteration is *selecting* otherwise. The operations of a selecting iteration are charged to the following edges (called *charged edges*): Those edges that were processed and selected in Step 1.1 (and so not eliminated), and came from a cluster that remained active at the end of the iteration. We show below that there are at least $3m/(4 \log^{(2)}n \log^{(3)}n)$ charged edges per selecting iteration (Claim 3). Also, we show that there are fewer than $3m/d$ charged edges over the course of the phase (Claim 4). Thus there are at most $\lfloor 4\log^{(2)}n \log^{(3)}n/d \rfloor = O(1)$ (for large enough n this is 0) selecting iterations per phase (since $d \geq (\log n \log^{(3)}n)^{1/2}$). Per phase, these iterations cost $O(\log n/\log^{(2)}n)$ time and $O(m/\log^{(2)}n \log^{(3)}n)$ operations. They are charged to the fixed budget. Over the whole algorithm this is $O(\log n)$ time and $O(m)$ operations.

So, over the whole algorithm, all three types of iterations perform $O(m)$ operations in $O(\log n \log^{(3)}n)$ time.

Reducing iterations.

Claim 1. There are only $O(\log^{(3)}n)$ reducing iterations in a phase.

Proof of Claim 1. After $\log^{(3)}n + \log^{(4)}n$ reducing iterations the number of active clusters decreases by a factor of at least $\log^{(2)}n \log^{(3)}n$. Thus, in the next iteration a processor will be assigned to each (presently) unprocessed block and following this iteration there will be no active clusters left. □

Eliminating iterations.

Claim 2. The number of edges eliminated in an eliminating iteration is at least $3m/(4\log^{(2)}n \log^{(3)}n)$.

Proof of Claim 2. Since at least half the clusters remain active, at least half the processors process blocks in clusters that remain active at the end of the iteration. Between them, these processors process at least $\frac{3m}{2 \log^{(2)} n \log^{(3)} n}$ edges. Since at least half of these edges are eliminated the claim follows. □

Selecting iterations

Claim 3. There are at least $3m/(4 \log^{(2)} n \log^{(3)} n)$ charged edges per iteration.

The proof of claim 3 is identical to that of Claim 2.

Claim 4. There are fewer than $3m/d$ charged edges over the course of the phase.

Proof of Claim 4. Consider a charged edge e and the iteration t in which it was charged. Let e be in a cluster of supervertex v. At the end of iteration t the cluster containing e remained active. Thus fewer than d actual edges incident on v had been discovered at the end of iteration t. We conclude that the last charged edge, in this phase, for supervertex v is at most the $d-1$st charged edge for supervertex v. As there are at most $3m/d^2$ supervertices in this phase the claim follows. □

It is clear that the complexity of the Step 1.2 is dominated by that of Step 1.1.

Step 2

As stated above, in the dth phase we apply the Shiloach/Vishkin connectivity algorithm for $O(\log d)$ time to the graph specified by the supervertices and the edges selected in step 1. This algorithm was described in Section 3.1. The SV connectivity algorithm requires that the edges be accessible via an array whose size is twice the number of the edges and each edge appears there once in each of its two directions. It is easy to achieve this. In Step 1 each of the (at most) m/d^2 growing supervertices selects at most d adjacent edges. So each supervertex (in Step 1.2) will place in this array its (at most) $2d$ entries, as follows. Each of its selected edges appears twice in this array, once in each of its directions. (Even if an edge was selected by its two endpoints, no damage will be incurred by the redundant work). Recall that the supervertices are of two types: growing and large. The large supervertices participate in this connectivity algorithm merely in a passive way. That is, they may be endpoints of edges, but no attempt is made to shortcut over such a supervertex; also, a large supervertex is never hooked to any other supervertex. We implement this as follows. When a star is hooked to a large supervertex, the supervertices comprising the star are all marked large and henceforth only participate in the connectivity algorithm in a passive way.

The algorithm termination theorem of Shiloach/Vishkin states that after applying $c \log n$ iterations of their basic ($O(1)$ time) step to any input graph, their algorithm must terminate, for some constant c. Recall that the spanning forest property, which is satisfied by the SV algorithm, implies that the algorithm actually provides a spanning tree for each supervertex formed in the course of the algorithm. This termination theorem implies the following.

Corollary. After $c \log d$ iterations, any (spanning) tree built by the SV algorithm must comprise at least d vertices.

Proof. Consider any tree built by the SV algorithm by the end of iteration $c \log d$. Suppose that this tree has less than d vertices. Consider the subgraph induced by the vertices of this tree. The SV algorithm has the following property (the description given in Sec. 3.1 above suffices to verify this): When applied to this subgraph, the algorithm may, after $c \log d$ iterations, arrive at exactly this tree, which contradicts the termination theorem. (Note that in the previous sentence we used the word "may" since because of the concurrent-write convention the SV algorithm is not deterministic and may provide different answers when applied to the same input). □

We deduce, from the corollary, that after $c \log d$ iterations, any tree built by the algorithm, comprising only growing supervertices, includes either at least d old supervertices or all the old supervertices, as claimed in observation (b) above (the only other possibility is that the tree contains a large supervertex).

Step 2 performs $O(m \log d / d)$ operations in $O(\log d)$ time (using the formulation of [SV-82], it is m/d processors and $O(\log d)$ time). This is charged to a miscellaneous budget. Over the whole algorithm this is $O(m)$ operations and $O(\log n)$ time.

Step 3

In the preceding Step 2 the old supervertices (whose number is $O(m/d^2)$) were combined into (larger) new supervertices (whose number is $O(m/d^3)$). Another change, which occurred in Step 1, is that edges were eliminated. Step 3 is responsible for reflecting these changes in the data structures of clusters, blocks and edges.

The input for Step 3 comprises:

(a) For each processor, the ordered list of the blocks it processed in Step 1.

(b) The array of the edges used in Step 2.

(c) The array of clusters. This array also includes dummy clusters, whose role will become clear later. The number of dummy clusters is less than twice the number of actual clusters.

(d) For each cluster, the array of its blocks from the start of Step 1.

Below we alternately use several patterns for the assignments of processors to jobs. One is called *the pattern of Step 1*: each processor processes one block at a time, in the same order as in step 1 (this is easily obtained from item (a) of the input to Step 3). The other patterns will be described as they are used.

Recall that each hooking in the SV algorithm is based on a causing edge. These edges play a central role in Step 3. Consider any new supervertex: the spanning forest property implies that its composing (old) supervertices together with causing edges form a tree. In order to simplify the exposition we neglected to add to Step 2 an instruction for marking each causing edge (immediately after it is used for hooking). So we assume that each causing edge is so marked. Whenever, in Step 3, we refer to a causing edge, we mean a causing edge introduced in Step 2 of the present phase.

Step 3.1. For each cluster we assign serial numbers, starting from one, to the causing edges in the cluster. This is done as follows. In Step 3.1.1 we assign a processor to each causing edge. In Step 3.1.2 we assign serial numbers to the causing edges in each block. In Step 3.1.3 we assign serial numbers to the causing edges of each cluster.

Step 3.1.1. We simply assign two processors to each edge used in Step 2, one processor for each direction of the edge (some of the processors may not be assigned a copy of a causing edge; such processors are idle during Step 3.1). Recall that $O(m/d)$ edges were used in Step 2.

Step 3.1.2. For each block, we compute both the number of its causing edges and the serial number of each of them. This is done, in the obvious way, using a balanced binary tree whose leaves correspond to the $O(\log n / \log\log n)$ edges of the block. The height of this tree is $O(\log\log n)$. This Step takes $O(\log\log n)$ time and $O(m \log\log n / d)$ operations ($O(m/d)$ edges multiplied by $O(\log\log n)$ time) per phase. This is charged to a miscellaneous budget. Over the whole algorithm, this is $O(\log n)$ time and $O(m)$ operations (remember that $d \geq (\log n \log^{(3)} n)^{1/2}$).

Step 3.1.3. For each cluster, we visit its blocks in turn (i.e., serially) and assign to each block a range for the serial numbers of its causing edges relative to the other causing edges of the cluster. This takes $O(\log^{(2)} n \log^{(3)} n)$ time and $O(m \log^{(2)} n / \log n)$ operations. This is charged to the fixed budget. Using the processor assignment pattern from Step 3.1.1 we assign serial numbers to each causing edge. This takes $O(1)$ time and $O(m/d)$ operations per phase. This is charged to a miscellaneous budget. Thus, over the whole algorithm, this step takes $O(\log n)$ time and $O(m)$ operations.

Step 3.2. Our goal in this step is to provide, for each new supervertex S, a (circular) linked list which goes through all the clusters in S. Recall that at the beginning of a phase the clusters of each (old) supervertex were arranged in a circular linked list. Let C be a cluster. Suppose C has h causing edges. In this circular list of clusters, we replace such a cluster C by a list comprising the cluster plus h dummy clusters; each of the dummy clusters represents one of the h causing edges (recall that these causing edges were introduced in the present phase). This still gives a circular list for each old supervertex.

Claim 5. The number of dummy clusters created during the whole algorithm is bounded by $2(n-1)$ which is $O(m/\log n \log^{(3)}n)$.

Proof of Claim 5. Recall that the input vertices and causing edges form a spanning forest; thus there are at most $n-1$ causing edges. □

Next, we show how to form a single circular list for each new supervertex. This new list will include all the clusters from the old supervertices that form the new supervertex. For this we apply an idea of [AV-84] for "stitching" the circular lists at the causing edges. Specifically, for each causing edge we do the following. A causing edge has a copy (which is a dummy cluster) in two of these circular lists. Each copy has a successor in its own list. In parallel, we *make the successor of each copy of each causing edge the successor of the other copy of the same causing edge*. An argument similar to [AV-84] shows that this indeed gives a single circular list for each new supervertex.

The number of operations required is proportional to the number of causing edges (or rather the number of edges used in Step 2), which is $O(m/d)$. The time is $O(1)$. This is charged to a miscellaneous budget. Over the whole algorithm this is $O(m)$ operations and $O(\log n)$ time.

It is convenient to place the dummy clusters introduced in this phase into the array of clusters. To do this we need to assign a serial number to each such dummy cluster; this is readily computed by means of a prefix sum computation with respect to the array of edges used in Step 2. Per phase this takes $O(\log n/\log^{(2)}n)$ time and $O(m/d)$ operations. This is charged to a miscellaneous budget. Over the whole algorithm this is $O(m)$ operations and $O(\log n)$ time.

Henceforth, when we refer to dummy clusters, we intend all the dummy clusters that are present, and not just those created in the current phase (see Step 3.5 for a discussion of why dummy clusters from previous phases might be present).

Step 3.3. The goal is to remove those edges eliminated by the edge selection in Step 1 and to form new blocks so that there is at most one incomplete block per cluster. We recall that an edge can only be eliminated from its cluster; it never moves to another cluster.

Step 3.3.1. For each block processed in Step 1 we remove those edges eliminated in Step 1, forming a list of the remaining edges. Also, for each block, we record the number of edges still present. The processor allocation for this Step is given by the pattern of Step 1; its complexity is dominated by that of Step 1.

Step 3.3.2. For each cluster we form a list of its blocks. We partition each such list into two sublists. The first sublist comprises the non-empty blocks that were processed in Step 1 and incomplete blocks (there is at most one incomplete block for each cluster). The second sublist contains the complete blocks that were not processed in Step 1. Empty blocks are discarded. This separation is performed in parallel for each cluster by a sequential scan of the list of blocks in the cluster. Next, for each cluster, for each first list, we compute the prefix sums of the block sizes. This step uses $O(\log^{(2)}n \log^{(3)}n)$ time and $O(m \log^{(2)}n/\log n)$ operations, per phase. This is charged to the fixed budget for the phase. Over the whole algorithm, this is $O(\log n)$ time and $O(m)$ operations.

Step 3.3.3. For each sublist of the first type, we form new blocks, as follows. Using the prefix sums, we determine the new block boundaries. (This requires scanning the edges in the blocks containing such boundaries.) Then we append each list of edges to the end of the list for the preceding block. For each cluster, the edges are now divided into complete blocks plus at most one incomplete block.

The processor allocation for this step follows the pattern of Step 1, except that in addition we need to process each old incomplete block, of which there is one per cluster; but this just requires an additional allocation of one processor per cluster, which is straightforward. Thus the complexity of this step is dominated by that of Step 1.

Step 3.3.4. For each new block, it remains to place its header in the block array for its cluster, and to place its edges in an edge array. For each new block B, we choose an old block B' whose place it takes; that is B's header will take the place occupied by B''s header, and B's edges will be placed in the edge array associated with B'. B' can be chosen according to the following rule: it is the first old block which overlaps with B. The complexity of the step so far is the same as that of Step 3.3.3.

Finally, for each cluster, we compress its block headers to the start of the block array, by means of a sequential scan. This takes $O(\log^{(2)}n \log^{(3)}n)$ time and $O(m \log^{(2)}n/ \log n)$ operations, per phase. This is charged to the fixed budget for the phase. Over the whole algorithm, this is $O(\log n)$ time and $O(m)$ operations.

Step 3.4. Our goal is to separate new growing from new large supervertices. Recall the lists of clusters constructed in Step 3.2. Let (u,v) be a causing edge. Let edge (u,v), in the direction from u to v, be represented by a dummy cluster in one of these lists.

(a) We attach the ordered pair (u,v) to this element.

Let C be the *ith* cluster of a vertex w.

(b) We attach the ordered pair $(|V| + i,w)$ to the element representing C in these lists.

Consider the lexicographic order on these pairs. The pairs were designed so that initially each element (in these lists of clusters) gets a different pair. We defined a large supervertex to be one where the number of incident edges is larger than $d^{4.5}$. Let $\gamma_d = \frac{d^{4.5}}{\log n \log^{(3)} n}$. Recall we assumed that each cluster contained exactly $\log n \log^{(3)} n$ edges initially. Thus, for each new growing supervertex, the circular list of clusters and causing edges comprises fewer than $3\gamma_d$ nodes (at most γ_d clusters and fewer than $2\gamma_d$ dummy clusters, corresponding to fewer than γ_d causing edges). It is convenient to redefine a large (resp. growing) new supervertex to be one with at least (resp. fewer than) $4\gamma_d$ nodes in the circular list of clusters and dummy clusters. This implies that a new large supervertex has at least $4/3\ d^{4.5}$ incident edges (since at most 2/3 of the nodes are dummy clusters), while a new growing supervertex has fewer than $4d^{4.5}$ incident edges.) For each of our lists we determine whether it represents a large or growing new supervertex by the following computation.

(c) We iterate at each element, in parallel, the following basic (doubling) operation $\beta_d = \log (2\gamma_d)$ times.

1. The new pair of the element is the minimum between its own pair and the pair of its successor.
2. Doubling.

Following this computation each element holds the minimal pair among its own (original) pair and the (original) pair of its $2^{\beta_d} - 1$ original successors.

(d) We check at each element whether its minimal pair and the minimal pair of its (present) successor are equal.

We observe that the cyclicity of each list implies that if the minimal pairs of some element and its present successor are equal then: (i) this minimal pair must be the minimal pair of their list. (ii) The list contains at most $2^{\beta_d+1} - 1$ elements and therefore the supervertex is growing. The converse also holds. That is, if a list contains at most $2^{\beta_d + 1} - 1$ elements then the list must have an element for which this equality holds.

(e) We apply β_d+1 parallel doublings in order to "broadcast" this minimum.

In each list, in which no minimal pair was found, nothing is being broadcast and each node in such a list can conclude that it is part of a new large supervertex. Next we give serial numbers, starting from one, to the clusters of each growing supervertex.

(f) Using β_d+1 parallel doublings, for the lists of growing supervertices, we rank each cluster with respect to the element whose pair is minimum in its list.

(g) Using β_d+1 parallel doublings, for the lists of growing supervertices, we shortcut over, and thereby discard, the dummy clusters.

The processor allocation for Step 3.4 is to provide one processor to each cluster and dummy cluster ($O(m/(\log n \log^{(3)} n))$ processors). Thus Step 3.4 takes $O(\log d)$ time and $O\left(\frac{m \log d}{\log n \log^{(3)} n}\right)$ operations. This is charged to the increasing budget for the phase. Over the whole algorithm, this is $O(\log n)$ time and $O(m)$ operations.

Step 3.5. Each new supervertex is presently represented by a circular list of clusters; the lists for large supervertices include dummy clusters, while the lists for growing supervertices do not (Step 3.4). During this step, the array of cluster headers is reordered so that clusters belonging to the same new growing supervertex are in contiguous locations. We remind the reader that each cluster comprises a set of complete blocks plus at most one incomplete block (Step 3.3).

We compute the new location for each cluster header as follows. >From part (f) of Step 3.4, we have already determined, for each new growing supervertex, the number of clusters it contains. This number is placed in the 'first' cluster for the supervertex; every other cluster in a new growing supervertex is assigned the number zero. The clusters in new large supervertices are all assigned the number one. By performing a prefix sum computation over these numbers with respect to the array of clusters, we obtain the new location of each first cluster of a new growing supervertex and of each cluster of a new large supervertex. Finally, each cluster of a new growing supervertex computes its location relative to the first cluster its new growing supervertex. This step uses $O(\log n/\log^{(2)} n)$ time and $O(m/\log n \log^{(3)} n)$ operations per phase. This is charged to the fixed budget for the phase. Over the whole algorithm it uses $O(\log n)$ time and $O(m)$ operations.

Analysis. We have shown that the complexity of each of the Steps, over the whole algorithm, is dominated by the sum of two components: the complexity of Step 1, and a complexity of $O(m)$ operations and time $O(\log n)$. Hence Step 3 performs $O(m)$ operations in $O(\log n \log^{(3)} n)$ time.

We conclude that,

Theorem 3.1: For $m \geq \log n \log^{(3)} n$ there is an optimal connectivity algorithm in the CRCW PRAM model; it performs $O(m + n)$ operations in time $O(\log n \log^{(3)} n)$.

Implementation of the Array ACTUAL

We show how to implement array *ACTUAL* in $O(mn^\epsilon)$ space, for any fixed $\epsilon > 0$. Since an alternative implementation in $O(n^2)$ space is obvious, we conclude that array *ACTUAL* can be implemented in $O(\min(n^2, mn^\epsilon))$ space. Our description below ignores the timestamps attached to each edge. The remark at the end takes care of this issue.

We use two arrays A and B. Array A is of size $n \times n^\epsilon$ and array B is of size $m \times n^\epsilon$. Suppose the i-th edge of the input graph connects supervertices u and v, where $1 \leq i \leq m$. We show how to store this edge in these arrays using a single processor. This parallel procedure will take at most $1/\epsilon$ steps. Each of u and v is a number between 1 and n. Suppose, without loss of generality, that (the number) u is smaller than (the number) v. Let $v_1, v_2 \cdots v_{1/\epsilon}$ be the representation of v with respect to base n^ϵ. (If $1/\epsilon$ is not an integer take instead $\lceil 1/\epsilon \rceil$.)

Step 1. Write i in location $A(u, v_1)$. Observe that concurrent writes may occur. Suppose that edge j_1 was actually written into $A(u, v_1)$. If $i = j_1$ then we have finished storing edge i. Otherwise,

Step l, $1 < l \leq 1/\epsilon$. Write i in location $B(j_{l-1}, v_l)$. Again, concurrent writes may occur. (The following is not required in Step $1/\epsilon$). Suppose that edge j_l was actually written into $B(j_{l-1}, v_l)$. If $i = j_l$ then we have finished storing edge i. Otherwise, proceed to Step $l+1$.

It is easy to verify that all m edges will be stored by the end of Step $1/\epsilon$. Also, given two vertices u and v, it takes at most $1/\epsilon$ steps to find out whether there exists an edge connecting u and v, using the information in arrays A and B.

Remark. So far, our description completely neglected the fact that we use the timestamps attached to each edge; to use the timestamps the storing procedure simply ignores values in arrays A and B from previous phases.

3.3. Connectivity in Logarithmic Time and Optimal Speed-up

Here we show how to implement the connectivity algorithm so that it runs in time $O(\log n)$. As in Subsection 3.2, we assume that $m \geq n \log n \log^{(3)} n$. We also assume that the number of processors available is $p = m/\log n$. The basic structure of the algorithm remains unchanged. The algorithm consists of $O(\log\log n)$ phases each with the same input/output definition as before. Steps 2 and 3 will also remain essentially unchanged. The only step we need to redesign is Step 1. This subsection is devoted to describing this new design.

We reduce the *size of a block* to $2\log n/(\log^{(2)} n \log^{(3)} n)$ edges. A cluster still comprises $\log^{(2)} n \log^{(3)} n$ blocks. Thus we have more clusters, by a factor of $1/2 \log^{(3)} n$. This requires us to reanalyze Step 3. However, a careful examination of the analysis we gave shows that it still uses $O(m)$ operations and $O(\log n)$ time over the whole algorithm.

Consider the execution of Step 1 in some phase of the algorithm of Sec. 3.2. Suppose this execution takes β iterations. Here, we perform Step 1 by a related method. Roughly, the iterations of each phase will be grouped in *clumps*; each clump comprises $O(\log^{(3)} n)$ iterations. (If $\beta < \log^{(3)} n$ this implies that we perform too many iterations for this phase; however, this will not matter.) Within each such clump the rescheduling of processors is performed with the aid of the redistribution procedure of Part I of this pair of papers (recall that the present paper is Part II). The reader is assumed to be familiar with this procedure. On each iteration, we aim to provide each processor with a distinct block. In an iteration, a processor processes one block, as before (assuming it has been allocated a block).

We now describe the scheduling. The processors are divided into sets of size $\log^{(2)} n \log^{(3)} n$. With each set we associate a *collection* of active clusters. The active clusters play the role of objects in the redistribution procedure. For each set of processors one processor is designated to participate in the redistribution procedure. Initially, the active clusters are evenly distributed among the collections; so each collection contains at most $\log^{(2)} n \log^{(3)} n$ clusters.

We need to redefine the notion of active for clusters. At the start of a clump of iterations a cluster is active according to the previous definition. A cluster can become *suspended*, and cease to be active, for either of the following two reasons:

(i) All its edges have been processed.

(ii) $\log n/(\log^{(2)} n \log^{(3)} n)$ actual outedges have been found from that cluster in the present clump.

A suspended cluster is no longer active for the duration of the clump. At the end of a clump the active and suspended clusters are reclassified as active or inactive according to the previous definition. (We remark, by way of example, that a suspended cluster could become active in the following situation. If $d > \log n/\log^{(2)} n \log^{(3)} n$, if for some supervertex v we suspend a cluster C because it has provided $\log n/\log^{(2)} n \log^{(3)} n$ actual outedges, and if the other clusters for supervertex v have been exhausted without providing actual outedges, then C will become active anew at the end of the clump.)

We remind the reader of several definitions from Part I. The active clusters are divided into collections. The *size* of a collection is the number of clusters it contains, and the *weight* of a collection is the square of its size. The collections are grouped in classes; class S_i comprises those collections whose size is in the range $(2^i, 2^{i+1})$, for $i > 0$, and S_0 comprises the collections of size 0 and 1. Let L_i denote $\bigcup_{k \geq i} S_i$. $|S_i|$ (resp. $|L_i|$) denotes the number of collections in S_i (resp. L_i). $wt(S_i)$ (resp. $wt(L_i)$) denotes the sum of the weights of the collections in S_i (resp. L_i). Let c_A be the number of active clusters and let q be the number of collections (these correspond to r, the number of objects and p, the number of processors, in Part I). On the average, there are c_A/q active clusters in each one of the q collections. Define the variable *average* to be $\lfloor \log c_A/q \rfloor$. Let α denote the smallest $i > average + 2$ such that $|S_i| > 1/32\ |S_{i-1}|$. *deg* denotes the degree of the expander graph used for the redistribution procedure (in Part I, *deg* was denoted by d). g and f are constants, $g = 298$ and $f = \max\{g, (8deg)^2\}$. Let W be the weight of the current set of collections, and let $WMIN$ be the minimum possible weight for c_A clusters (recall that c_A is the size of the current set of active clusters); note that $q\ \lfloor c_A/q \rfloor^2 \leq WMIN \leq q\ \lceil c_A/q^2$. The collections are *balanced* if W is bounded by either fq or $g\ WMIN$.

A clump comprises $h\log^{(3)}n$ iterations (where h is some proper constant specified later); it proceeds as follows.

(i) For each collection, allocate the processors evenly to the clusters. Each processor processes one block from its allocated cluster (in so far as blocks are available). Then, for each cluster, determine if it is still active and count the number of active clusters in each collection. This step uses $O(m/(\log^{(2)}n \log^{(3)}n))$ operations and $O(\log n/\log^{(2)}n \log^{(3)}n)$ time.

(ii) Perform one step of the redistribution procedure. This takes $O(1)$ time and $O(m/(\log n \log^{(2)}n \log^{(3)}n))$ operations.

If following a clump there remain at most $g_1 m/(\log n \log^{(2)}n \log^{(3)}n)$ active clusters, as opposed to suspended and inactive clusters, (for some proper constant g_1 characterized later) we call it a *reducing* clump. If the number of edges eliminated during a clump is proportional to the number of operations performed during the clump, then we call it an *eliminating* clump. Specifically, as the number of operations in a clump is $O(m/(\log n \log^{(2)}n))$, a clump is eliminating if the number of edges eliminated during the clump is at least

$g_2 m/(\log n \log^{(2)} n)$ (for some proper constant g_2 characterized later). Before proceeding to the description of the new Step 1, we formulate the main lemma of our complexity analysis. This lemma motivated the design of the new Step 1.

Main Lemma. (1) Each clump is either reducing or eliminating (or both).

(2) After a reducing clump, the number of clusters associated with active supervertices is $O(m/(\log n \log^{(2)} n \log^{(3)} n)$.

Item (1) of the Main Lemma guarantees that if a clump is not reducing then we advance by means of eliminating "enough" edges. Item (2) shows that after a (single) reducing clump, the number of clusters belonging to still active supervertices must be so small that we can finish processing all the blocks in these clusters in one additional iteration.

We are ready to describe the new Step 1.

The new Step 1.

Perform a clump. Following the clump, in $O(\log n/\log^{(2)} n)$ time, determine if the clump was reducing or eliminating. Continue performing clumps until a reducing clump is performed.

Then, in one more iteration finish Step 1 of the present phase.

We first prove two lemmas and then proceed to prove the Main Lemma.

Lemma 3.4.1. There are at most $h_1 \log^{(3)} n$ iterations in a clump for which the collections are not balanced following Step (i), for some constant h_1.

Remark. We will select h to be larger than h_1. (Recall that a clump comprises $h \log^{(3)} n$ iterations.)

Important Remark. In fact, these $h_1 \log^{(3)} n$ iterations must include all iterations for which the collections do not obey one of the conditions below, at the end of Step (i).

Condition 1. Either α exists and $wt(L_\alpha) \leq 1/8\ W$, or α does not exist. (If this condition is obeyed then, by Lemmas 4.3 and 4.4 of Part I, $W \leq gWMIN$ and the collections are balanced).

Condition 2. The total weight of the collections is bounded by $\dfrac{3fm}{\log n \log^{(2)} n \log^{(3)} n}$. (If this condition is obeyed then $W \leq fq$ and the collections are balanced).

Proof of Lemma 3.4.1. As in the proof of Lemma 3.1, Part I: The weight before a clump starts is at most $\dfrac{3m}{\log n \log^{(2)} n \log^{(3)} n} (\log^{(2)} n \log^{(3)} n)^2$. If during the clump the weight goes below $\dfrac{3fm}{\log n \log^{(2)} n \log^{(3)} n}$, then the collections are balanced (by definition). The ratio of these two weights is $O((\log^{(2)}n \log^{(3)}n)^2)$. Given an iteration in which the collections are not balanced after Step (i), it can be shown that the weight is reduced by a constant multiplicative factor in Step (ii) (see Theorem 3.1 of Part I for the details). In fact, this proof shows that if α exists and Condition 2 does not hold then the weight reduction is proportional to $wt(L_\alpha)$. Thus if neither Condition 1 nor Condition 2 holds then the total weight is reduced by a constant multiplicative factor. Therefore, after $O(\log^{(3)}n)$ such iterations the initial weight is reduced to at most $\frac{3fm}{\log n \log^{(2)}n \log^{(3)}n}$ and the lemma follows. □

Lemma 3.4.2. If either α exists and $wt(L_\alpha) \leq 1/8\, W$, or α does not exist (i.e., Condition 1 holds), and in addition Condition 2 does not hold, then there is a class S_k such that $wt(S_k) \geq W/8$ and at least $1/2^9$ of the collections are in S_k.

Proof. First, we show that there is a class S_k with $wt(S_k) \geq W/8$, $average - 1 \leq k \leq average + 2$. We start by showing that $wt(L_{average+3}) \leq wt(S_{average+2}) + W/8$. This follows because for $average + 3 \leq i < \alpha$, $wt(S_{i-1}) \geq 2\, wt(S_i)$, by the definition of α (we note that the result holds even if α does not exist). Next, we observe that $\sum_{i \leq average-2} wt(S_i) \leq W/4$. Thus $wt(S_{average+2}) + \sum_{-1 \leq i \leq 2} wt(S_{average+i}) \geq 5W/8$; one of these four classes must have weight at least $W/8$; let this class be S_k.

Now, we determine a lower bound on the number of collections in S_k. The smallest number of collections are present if $k = \alpha$; in this case, since $k = average + 2$, each collection in S_k has weight at most 4^3 times larger than the average weight of a collection; so at least $1/8 \cdot 1/4^3 = 1/2^9$ of the collections are in S_k. □

Proof of item (1) of the Main Lemma. Consider some given clump.

Suppose that Condition 2 is satisfied when the clump is over. This implies that there are $O(m/(\log n \log^{(2)}n \log^{(3)}n))$ active (as opposed to suspended or inactive) clusters remaining. We conclude that this is a reducing clump.

Suppose Condition 2 is not satisfied and Condition 1 is satisfied following Step (i) of an iteration of the clump. By Lemma 3.4.2, S_k exists. Consider a collection that is in S_k at the end of Step (i) of this iteration. Such a collection satisfies at least one of the following two conditions:

Condition A. It was in $S_{k+1} \cup S_k$ at the start of Step (i) and it lost at most half of its clusters (through their becoming inactive).

Condition B. It was in $\bigcup_{i>k} S_i$ at the start of Step (i) and it lost at least half of its clusters.

In order to complete the proof of the Lemma we identify two cases.

Case 1: At least half the collections in S_k obey condition A. We show below that,

Claim 1: In case 1, the number of edges eliminated is proportional to the number of operations performed in Step (i).

Case 2: At least half the collections in S_k obey condition B. We show below that,

Claim 2: In case 2, the weight of the collections decreases by a constant multiplicative factor.

As in the proof of Lemma 3.4.1, Claim 2 implies that there can be at most $h_2 \log^{(3)} n$ iterations for which case 2 holds, for some constant h_2. We first finish the proof of item (1) of the Main Lemma and later prove Claims 1 and 2. Select h to be $2 \cdot \max(h_1, h_2)$ and assume that we are given a clump which is not reducing. Then, at least $h/2$ of the iterations fall into Case 1, above. We deduce that the number the edges eliminated during the clump is proportional to the number of operations performed during the clump, and therefore the clump is eliminating. Item (1) of the Main Lemma follows.

Proof of Claim 1. Observe that for each cluster that remained active, at the end of Step (i) of an iteration, for each of its blocks that was processed, at least half the edges processed during Step (i) were eliminated. For a given collection, we tried to process b or $b+1$ blocks from each cluster in the collection in Step (i) (it may be that a cluster had fewer than $b+1$ blocks). Thus, for a collection that lost at most half its clusters (i.e. obeying Condition A), at least $\frac{b}{2b+1} \geq \frac{1}{3}$ of the blocks processed lost at least half their edges. By Lemma 3.4.2, at least $1/2^9$ of the collections are in S_k, and by Case 1 at least 1/2 of these collections obey Condition A, that is at least $1/2^{10}$ of all the collections; by the previous sentence for each of these (at least $1/2^{10}$) of the collections, at least 1/3 of the processors associated with these collections (i.e. at least $1/(3 \cdot 2^{10})$ of the processors) eliminated at least 1/2 of the edges they processed. We deduce that the number of edges eliminated by the processors is proportional to the total number of operations performed.

Proof of Claim 2. Each collection that lost at least half its clusters lost at least 3/4 of its weight. Thus if W is the current weight of all the collections (i.e. the weight after Step (i)), then the total weight at the start of Step (i) was at least $W(1 + 3 \cdot 1/8 \cdot 1/5)$. This can be seen as follows. the clusters in S_k comprise at least 1/8 of the total weight (Lemma 3.4.2). The collections in S_k have weight ranging from 2^{2k} to $2^{2(k+1)}$, so the collections obeying

Condition B (at least 1/2 of the collections in S_k) comprise at least 1/5 of the collections in S_k, by weight. Thus the collections obeying Condition B comprise at least $1/8 \cdot 1/5$ of the current weight; since these clusters each lost at least 3/4 of their weight in Step (i), the total weight at the start of Step (i) was at least $W(1 + 3 \cdot 1/8 \cdot 1/5)$.) Hence the current weight is at most $\frac{1}{1 + 3 \cdot 1/8 \cdot 1/5}$ of the weight at the start of Step (i). □

Proof of item (2) of the Main Lemma: Consider an active supervertex. It has fewer than $d \log^{(2)}n \log^{(3)}n / \log n$ suspended clusters that contain unchecked edges (this is 0 for $d < \log n/(\log^{(2)}n \log^{(3)}n)$). (This follows because $\log n/\log^{(2)}n \cdot \log^{(3)}n$ actual outedges were found for each suspended cluster that was not exhausted.) But there are at most $3m/d^2$ growing supervertices in the current phase. Thus the number of suspended clusters associated with active supervertices is bounded by $3m \log^{(2)}n \log^{(3)}n / (d \log n)$, which is bounded by $3m\left(\frac{\log^{(2)}n \log^{(3)}n}{\log n}\right)^2$ which is $O(m/(\log n \log^{(2)}n \log^{(3)}n))$. Since $O(m/(\log n \log^{(2)}n \log^{(3)}n))$ is also the number of active clusters after a reducing clump, the lemma follows. □

Summary of the complexity analysis. By item (2) of the Main Lemma, the processing following a reducing clump uses $O(m/(\log^{(2)}n \log^{(3)}n))$ operations and $O(\log n/\log^{(2)}n)$ time. Over the whole algorithm this is $O(m)$ operations and $O(\log n)$ time.

A clump performs $O(m/\log^{(2)}n)$ operations in $O(\log n/\log^{(2)}n)$ time. Over the course of the algorithm there are $O(\log^{(2)}n)$ reducing clumps. Since an eliminating clump is responsible for eliminating $\Omega(m/\log^{(2)}n)$ edges, we conclude that there are only $O(\log^{(2)}n)$ eliminating iterations. Thus Step 1, over the whole algorithm, performs $O(m)$ operations in $O(\log n)$ time.

We have shown

Theorem 3.2: There is a CRCW PRAM algorithm for computing the connected components of a graph. It performs $O(m + n)$ operations in $O(\log n)$ time, if $m \geq n \log n \log^{(3)}n$.

3.4. The Reduction Procedure

We show how to reduce the connectivity problem for the case $m < n \log n \log^{(3)}n$ to the case $m \geq n \log n \log^{(3)}n$. The reduction procedure will build supervertices each of which has at least d^2 incident edges (from the input graph) with $d = \log n / \alpha(m, n) \log^{(2)}n$. (Henceforth we will use α to denote $\alpha(m, n)$.) Following the application of the reduction procedure we rename all the edges with their current supervertex endpoints, obtaining a *reduced* graph. We can then apply the connectivity algorithm of Section 3.2 to the reduced graph; the supervertices of the reduced graph are the vertices for the connectivity algorithm. (This ensures that for the reduced graph the number of vertices is bounded by $m/(\log n \log^{(3)} n)$. As in Section 3.2, by adding at most $m/2$ self loops, we guarantee that $\log n \log^{(3)} n$ is a lower bound to the vertex degree, and that the degree of each vertex is an integer multiple of $\log n \log^{(3)} n$).)

The reduction procedure uses the same basic procedure as the connectivity algorithm, that is we have a series of phases parameterized by d. Rather than have d increase to $d^{1.5}$, from one phase to the next, we increase d to the smallest power of two which is at least $d^{1.5}$. Initially $d = d_0$ (a constant, specified in the proof of the Lemma at the end of the present section); this is justified by introducing up to $1/2\ d_0^2 n$ self loops. During the algorithm we will introduce up to $1/2\ (2 + d_0^2)m$ further self loops. (So altogether we may add at most $(1 + d_0^2) \cdot \max(n,m)$ self loops. This is proven in the Lemma at the end of this section. Below, we will assume that n is bounded by m. We leave it to the reader to verify that the analysis carries through even if this assumption does not hold.) The self loops will ensure that all the blocks and clusters built during the algorithm are complete.

Now we proceed through only $O(\log^{(3)} n)$ phases. Each phase is divided into 3 steps, as before. The role of each step is identical. In fact, Step 2 is implemented just as before. Steps 1 and 3 need to be modified. We describe the changes to each step in turn.

Step 1.

Here, the edges are organized in blocks and clusters, as before. A cluster will contain at most $d \log d$ edges (counting eliminated edges, it will contain exactly this number of edges); each cluster is divided into at most $\log d$ blocks of d edges each (as in the main procedure, due to self loops one block per cluster may contain one additional edge). The cluster headers are kept in an array, clusters belonging to the same growing supervertex being contiguous. The block headers in each cluster are kept in an array of size $\log d$. The edges in each block are stored at the leaves of a depth $\log d$ binary tree. The nodes for this tree are provided by associating two nodes with each edge; the fact that d is a power of two implies that for each block, we have exactly one spare node. To enable rapid access from one edge to the next, we also keep the edges in a linked list. This additional structure for the blocks is needed in Step 3; we show in Step 3 how to maintain this structure. Recall that in the main procedure no edges were added to any of the clusters. Here, however, the clusters keep growing. New clusters will be formed (roughly speaking) by a union of old clusters. It will still be impossible for two edges that belong to the same cluster at some time in the algorithm to belong later to different clusters. We also keep an array of supervertices. Let $c = 2(d_0^2 + 2)$; we have at most cm incident edges, counting each edge twice, once for each endpoint. Thus we will have at most cm/d^2 supervertices, $cm/d \log d$ clusters, and cm/d blocks.

Basically Step 1 proceeds as above. However, we now assign $\frac{\log n}{\alpha\, d \log^{(2)} n}$ blocks to each processor. Another important change is that an operation $x := T(v)$ will take time $O(\alpha)$ rather than $O(1)$. (See the Appendix for a description of how to maintain the vertex table).

We divide the computation of Step 1.1 into two parts for the purposes of the analysis. The first part is Step 1.1.1. This part is most easily understood if we assume that $\frac{cm}{d \log d}$ processors are available, we simply simulate this by the actual number of processors which is only $\left(\frac{c\, m\, \alpha}{\log n}\right)$; so $\frac{cm}{d \log d}$ blocks are processed in each iteration. Thus for each active cluster at least one block is processed in each iteration. Step 1.1.1 requires $O(d\alpha)$ time and performs $O\left(\frac{m\alpha}{\log d}\right)$ operations. The second, and less interesting part, is Step 1.1.2 (determining if a supervertex is still active). Per iteration, this requires time $O\left(\log n/ \log^{(2)} n\right)$ and $O\left(\frac{m}{d \log d}\right)$ operations. So an iteration (of steps 1.1.1 and 1.1.2) uses $O\left(\frac{m\alpha}{\log d}\right)$ operations and $O(\log n/\log^{(2)} n)$ time.

As before, we divide the iterations into reducing, eliminating, and selecting iterations.

There are only $O(\log^{(2)} d)$ reducing iterations per phase (the reasoning being as before: After $\log^{(2)} d$ reducing iterations there is a processor standing by each block of each active cluster). The reducing iterations, per phase, require $O\left(\frac{m\, \alpha \log^{(2)} d}{\log d}\right)$ operations and $O\left(\frac{\log n \log^{(2)} d}{\log^{(2)} n}\right)$ time. Thus, over the whole procedure, the reducing iterations perform $O(m\alpha)$ operations in $O(\log n)$ time.

An eliminating iteration eliminates $\Theta(m/\log d)$ edges using $\Theta\left(\frac{m\, \alpha}{\log d}\right)$ operations and $O\left(\log n/\log^{(2)} n\right)$ time. Thus, over the whole algorithm, the eliminating iterations perform $O(m\, \alpha)$ operations in $O(\log n)$ time.

A selecting iteration will charge its operations to the following *charged edges*: those edges that were processed but not eliminated, and came from a cluster that remained active at the end of the iteration. In an iteration there are $\Omega(m/\log d)$ charged edges. Also, at most $O(m/d)$ edges can be charged during the phase. Thus there are $O(1)$ selecting iterations over the whole algorithm. They perform $O(m\ \alpha)$ operations in $O(\log n)$ time.

Step 1.2. For each growing supervertex, among the edges selected in Step 1.1. select d (or all, if $< d$ are available). The selected edges are placed in an array of size $O(m/d)$ (this implies we assign serial numbers to the edges, which is done using a parallel prefix sum computation with respect to the cluster headers). Again, the complexity of Step 1.2 is dominated by the complexity of Step 1.1.

Step 3. Many of the substeps are similar to those for the main algorithm. Steps 3.1-3.5 have the same goals as before. In addition, we add Step 3.6 to prepare new larger blocks and clusters for the next phase, and we add Step 3.7 to maintain the vertex table.

Step 3.1. This step is identical. However, the analysis of Step 3.1.2 changes. It now uses $O(\log d)$ time and $O(m \log d/ d)$ operations per phase. Likewise, Step 3.1.3 now uses $O(\log d)$ time and $O(m/ d)$ operations. Over the whole algorithm this is $O(\log n)$ time and $O(m)$ operations.

Step 3.2. This step is essentially identical. However, for each dummy cluster we introduce, we add $d \log d$ self loops (counting each self loop twice, once for each endpoint); this is a total of at most $2cm\log d /d$ self loops per phase. The complexity per phase is $O(\log n/\log^{(2)}n)$ time and $O(m/d)$ operations. Over the whole algorithm this is $O(\log n)$ time and $O(m)$ operations.

Step 3.3. This step is essentially identical. There are two changes. First, for each new block, we place the edges at the leaves of a depth $\log d$ binary tree. Second, for each cluster, we introduce self loops so as to make complete the at most one incomplete block we create (recall that all the blocks present at the start of the phase were complete, thus the only incomplete blocks result from Step 1). This introduces at most $cm/ \log d$ edges per phase (counting each self loop twice, once for each endpoint). As before, the complexity of this step has two components; the first component is dominated by the complexity of Step 1, while the second becomes $O(\log d)$ time and $O(m/d)$ operations per phase. Over the whole algorithm, the second component totals $O(\log n)$ time and $O(m)$ operations.

Step 3.4. As before, the goal is to separate the growing and large supervertices. We use a different method here. In addition, because of the need to create larger clusters for the next phase, each large supervertex will be divided into *'pseudo' growing supervertices.* We need to introduce the pseudo growing vertices in order to guarantee that we halve the number of supervertices in the vertex table from phase to phase. We will still retain the structure of the large supervertices. This will be made precise later.

Suppose we are given a set of circular lists stored in an array of size h. A 2-ruling set is a subset U of the nodes which satisfies the following:
(1) For each node which is not in U either its successor or its predecessor in its list (or both) are in U.
(2) For each vertex which is in U its successor in its list is outside U.
Before describing the rest of the algorithm, it is useful to recall that there is a 2-ruling set algorithm that uses $O(\log h / \log^{(2)} h)$ time and $O(h)$ operations [CV-87].

We iterate the following computation $2 \log d - \log \log d$ times.
(i) Apply the 2-ruling set algorithm to the lists constructed in Step 3.2 and then shortcut over the nodes not placed in the 2-ruling set.
(ii) Compress the selected nodes to the start of the array.

(The initial array is a copy of the array of cluster headers.) Any list that is reduced to a single node is defined to represent a new growing supervertex. Note that a node of a 2-ruling set is followed, in its circular list, by one or two nodes which are not in the ruling set and then by a node of the 2-ruling set. Thus a list that is reduced to a single vertex contains at least d^3 incident edges and at most $d^{2\log_2 3} \le d^4$ incident edges (this follows because, first, such a list contains at least $2^{2\log d - \log\log d} = \frac{d^2}{\log d}$ nodes, and at most $3^{2\log d - \log\log d} \le \frac{d^{2\log_2 3}}{\log d}$ nodes; second, each cluster contains $d \log d$ edges, counting eliminated edges). For the remaining new supervertices, the new large supervertices, we define each sublist that has been reduced to a single node to be a new *pseudo growing supervertex.*

Per phase, the complexity of this step is $O(\log n \log d / \log^{(2)} n)$ time and $O(m/d)$ operations. Over the whole algorithm this is $O(\log n)$ time and $O(m)$ operations.

Step 3.5. This step is identical. We treat all the clusters as parts of growing supervertices (this includes the pseudo growing supervertices). Over the whole algorithm this step uses $O(\log n)$ time and $O(m)$ operations.

Step 3.6. Here, we create new larger clusters and blocks for the next phase.
Step 3.6.1. For each growing and pseudo growing supervertex, we combine sets of old clusters (each of $d \log d$ edges counting eliminated edges) to form new clusters (each of

1.5 $d^{1.5}\log d$ edges, except possibly for one incomplete cluster; actually, we recall that each value of d was required to be a power of 2, and adjust the parameters accordingly). By adding at most 1/2 1.5 $cm \log d / d^{1.5}$ self loops we make each incomplete cluster complete (this follows because there are at most cm/d^3 incomplete new clusters, one for each new growing or new pseudo growing supervertex; and for each incomplete cluster we add at most 1/2 1.5 $d^{1.5} \log d$ self loops). This step uses a prefix sum computation with respect to the old array of clusters in order to identify the new clusters. Per phase, the step has complexity $O(\log n/\log^{(2)}n)$ time and $O(m/ d \log d)$ operations. Over the whole algorithm, this is $O(\log n)$ time and $O(m)$ operations.

Step 3.6.2. For each new cluster we combine sets of $d^{.5}$ blocks to form new blocks of size $d^{1.5}$ (the comment from the previous paragraph regarding the size of d applies here too). Recall that the edges in each old block were stored at the leaves of a binary tree of depth $\log d$. The trees for the new blocks are obtained by 1/2 $\log d$ pairwise combinings of the trees for the old blocks. (Note that when combining the trees for two old blocks, the spare node from one old block provides the new root, while the spare node from the other old block provides the spare node for the new block.) Recall that all the old blocks are complete (through the previous additions of self loops). Per cluster, there will be at most one incomplete block, which is made complete by the addition of self loops. This requires the addition of at most $cm/ d^{1.5}$ edges, counting each edge twice. Per phase this step has complexity $O(\log d)$ time and $O(m \log d/d)$ operations. Over the whole algorithm, this is $O(\log n)$ time and $O(m)$ operations.

Step 3.7. In this step, we update the vertex table. The new growing supervertices (including pseudo growing supervertices) will be placed in an array, and each old supervertex is given the name of its new supervertex. We also guarantee that the number of new growing supervertices is at most half the number of old growing supervertices.

There is a difficulty with the new pseudo growing supervertices. The clusters of an old supervertex may be split over several new pseudo growing supervertices. Thus we identify each old supervertex (growing or pseudo growing) with its 'first' cluster; each old supervertex is defined to belong to the new supervertex (growing or pseudo growing) that contains its first cluster. This definition causes no difficulty: any vertex in a large supervertex S is identified with a pseudo growing supervertex that is a part of S. Identifying the new name of each old supervertex takes $O(1)$ time. It remains to place the new supervertices (growing or pseudo growing) in an array. This is achieved with the use of a prefix sum computation with respect to the old array of supervertices.

The number of supervertices is reduced by a factor of at least $1.5d^{.5} \geq 2$ per phase, as required. Per phase, the complexity of this step is $O(\log n/\log^{(2)}n)$ time and $O(m/d \log d)$ operations. Over the whole algorithm, this is $O(\log n)$ time and $O(m)$ operations.

Analysis: As before, the complexity of Step 3 is bounded by the complexity of Step 1, plus a complexity of $O(\log n)$ time and $O(m)$ operations.

Lemma: At most $cm/4$ self loops are added during all the phases of the algorithm.

Proof: Suppose inductively that we end the algorithm with at most cm incident edges, counting each edge twice. We show that at most $cm/2$ edges are added (counting each edge twice); since we started with at most $m + d_0^2 n < cm/2$ incident edges and self loops, we conclude that the inductive hypothesis was correct. This also proves the lemma.

Consider a single phase. We added at most $2cm \log d/ d$ edges in Step 3.2, at most $cm/\log d$ edges in Step 3.3, at most $1.5\, cm \log d/ d^{1.5}$ edges in Step 3.6.1, and at most $cm/ d^{1.5}$ edges in Step 3.6.2. So the total number of edges added is at most $\sum_{d \geq d_0} \frac{2cm \log d}{d} + \frac{cm}{\log d} + \frac{1.5\, cm \log d}{d^{1.5}} + \frac{cm}{d^{1.5}}$. Take $\log d_0 = 24$; the sum is then bounded by $\sum_{d \geq d_0} \frac{4cm}{\log d} \leq \frac{12cm}{\log d_0} \leq \frac{1}{2} c$ (remember $\log d$ grows by a factor of 1.5 per phase). The lemma follows. (We remark that this is a rather crude bound.) □

We conclude

Theorem 3.3. There exists a CRCW PRAM algorithm that reduces the general connectivity problem to a connectivity problem for a graph with at most $m/(\log n \log^{(3)}n)$ vertices and at most m edges; it performs $O((m + n)\alpha(m, n))$ operations in time $O(\log n)$. Hence there is a CRCW PRAM algorithm for connectivity with complexity $O((m + n)\alpha(m, n))$ operations and $O(\log n)$ time.

4. A CREW Connectivity Algorithm

As this result is very similar to the CRCW connectivity algorithm, rather than give the full algorithm, we just indicate the changes that have to be made. As before, the algorithm has two parts: a main algorithm for the case $m \geq n \log^2 n$, and a reduction procedure for the case $m < n \log^2 n$. We discuss each in turn.

The main algorithm will have $O(\log n)$ phases; the invariant for the dth stage is that there are at most $n_d = (3/4)^d n$ supervertices at hand. A phase is performed in 3 steps; each step plays the same role as before. In Step 1, we select one outedge each for at least half the supervertices; we call those supervertices for which an outedge was selected *participating* supervertices. In Step 2, each participating supervertex will be linked to some other supervertex by a hooking operation (we simply perform one step of the Shiloach/Vishkin algorithm). In Step 3, all the trees formed in Step 2 are reduced to stars. We elaborate on each step in turn.

Step 1. For each supervertex, we partition its edges into blocks of log n edges; clusters are not needed here. We store the block headers in an array, blocks belonging to the same supervertex being contiguous. Also, for each supervertex, we keep its block headers in a linked list. The edge selection is performed by a sequence of iterations. Each iteration uses $m/\log^2 n$ processors. In each iteration, an equal number of processors are assigned to each supervertex, and each processor processes a block, in so far as blocks are available. To process a block, we determine for each edge in the block whether it is redundant or an outedge (we do not need to detect duplicate outedges so we no longer need the array ACTUAL). At the end of the iteration, we determine, for each supervertex, whether an outedge was found. If for at least half the supervertices an outedge was found we proceed to Step 2; otherwise, another iteration is performed. Each iteration performs $O(m/\log n)$ operations in $O(\log n)$ time. An iteration is defined to be *reducing* if for at least half of the supervertices an outedge has been found, and to be *eliminating* otherwise. As for the CRCW algorithm, we can show that an eliminating iteration performs a number of operations proportional to the number of edges eliminated. Thus over the whole algorithm, the iterations take $O(m)$ operations and $O(\log^2 n)$ time.

Step 2. We note that we selected at most one outedge for each supervertex. This enables us to avoid the concurrent writes that occur in the Shiloach/Vishkin algorithm. We proceed as follows.

(1) For each participating supervertex v, if v is adjacent to a smaller numbered supervertex w, hook v to w.

(2) For each participating supervertex v that neither hooked nor was hooked onto in (1), perform a hooking.

Step 2 requires $O(n_d)$ operations and $O(\log n)$ time per phase (the time is used to allocate the processors to the supervertices); this is a total of $O(m)$ operations and $O(\log^2 n)$ time over the whole algorithm.

Step 3. We proceed as in the main CRCW algorithm. First, for each new supervertex we obtain a single linked list of its blocks, including dummy blocks due to the hooking edges (the dummy blocks here play the role of the dummy clusters in the main algorithm). Then, using the optimal list ranking algorithm of [CV-86c], we determine to which new supervertex each block belongs and we eliminate all the dummy blocks. Finally, we reorder the blocks. So Step 3 uses $O(m/\log n)$ operations and $O(\log n)$ time per phase, or a total of $O(m)$ operations and $O(\log^2 n)$ time over the whole algorithm. here.)

Thus the CREW connectivity algorithm, for $m \geq n\log^2 n$ performs $O(m)$ operations in $O(\log^2 n)$ time.

We turn to the reduction procedure. It uses $m\alpha/\log^2 n$ processors and $O(\log^2 n)$ time. It has the same structure as the main algorithm of this section. The block sizes will now depend on the phase number, however. In phase s, $12i \leq s < 12(i+1)$, we maintain blocks of 2^i edges. Since $(3/4)^3 < 1/2$, twelve phases reduce the number of supervertices by a factor of $(1/2)^i = 1/16$, and therefore we have at most $m/16^i$ supervertices at hand. We note that when $2^i = \log^{1/2} n$, there are at most $m/\log^2 n$ supervertices at hand, so throughout the reduction procedure $2^i < \log^{1/2} n$. Step 1 is changed in two ways. First, as in Section 3.4, performing an access to the vertex table takes $O(\alpha)$ time rather than $O(1)$ time. Second, in an iteration, we proceed as if $m/4^i$ processors were at hand, rather than $m\alpha/\log^2 n$. Now, an iteration requires $O(\alpha 2^i + \log n) = O(\log n)$ time and $O(m\alpha/2^i)$ operations. This increases the overall complexity of Step 1 to $O(m\alpha)$ operations but the running time is still bounded by $O(\log^2 n)$ time. Step 2 is as before. In Step 3 we proceed as before, except that every twelfth phase we need to combine blocks so as to increase their sizes. The complexity of Step 3, in phase s, is $O(m/2^i)$ operations and $O(\log n)$ time, where $12i \leq s < 12(i+1)$; this is a total of $O(m)$ operations and $O(\log^2 n)$ time over the whole algorithm. Thus, overall, the reduction procedure performs $O(m\alpha)$ operations in $O(\log^2 n)$ time.

Remark: We cluster the phases into sets of 12 for the following reasons. First, since in one phase, the number of supervertices is reduced by one quarter, to reduce the number of supervertices to $n/\log^2 n$ requires 6 log log n phases. Second, as the phases proceed, we need the number of blocks to decrease exponentially. So we choose to maintain blocks of size 2^i, for $i = 0, 1, 2, \ldots$, in turn. Third, we must ensure that a phase takes only $O(\log n)$ time (for otherwise eliminating iterations could cost more than $\log^2 n$ time). This is ensured by making $O(\alpha\, 2^i) = O(\log n)$, which certainly holds if $2^i = O(\log^{1/2} n)$. That is to say, we can increment i at most 1/2 log log n times. It follows that i should be incremented every 12 phases.

References

[At-84] M. Atallah, "Parallel strong orientation of an undirected graph", *Information Processing Letters* 18 (1984), 37-39.

[AIS-84] B. Awerbuch, A. Israeli and Y. Shiloach, "Finding Euler circuits in logarithmic parallel time", *Proc. Sixteenth Annual ACM Symp. on Theory of Computing*, 1984, 249-257.

[AS-83] B. Awerbuch and Y. Shiloach, "New connectivity and MSF algorithms for Ultracomputer and PRAM", *Proc. Int. Conf. on Parallel Processing*, 1983, 175-179.

[AV-84] M. Atallah and U. Vishkin, "Finding Euler tours in parallel", *J. Computer and Systems Sciences* 29 (1984), 330-337.

[B-74] R.P. Brent, "The parallel evaluation of general arithmetic expressions," *J. ACM* 21,2 (1974), 201-206.

[C-86] R. Cole, "An optimal parallel selection algorithm", TR 209, Dept. of Computer Science, Courant Institute, NYU, 1986.

[CLC-82] F.Y. Chin, J. Lam and I. Chen, " Efficient parallel algorithms for some graph problems", *Comm. ACM* 25 (1982), 659-665.

[CV-86a] R. Cole and U. Vishkin, "Deterministic coin tossing with applications to optimal parallel list ranking", *Information and Control* 70 (1986), 32-53.

[CV-86b] R. Cole and U. Vishkin, "The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time", TR 52/86, Dept. of Computer Science, Tel Aviv Univ., 1986; also, TR 242, Dept. of Computer Science, Courant Institute, NYU, 1986.

[CV-86c] R. Cole and U. Vishkin, "Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time", TR 54/86, Dept. of Computer Science, Tel Aviv Univ., 1986; also, TR 244, Dept. of Computer Science, Courant Institute, NYU, 1986.

[CV-86d] R. Cole and U. Vishkin, "Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms", *Proc. Eighteenth Annual ACM Symp. on Theory of Computing*, 1986, 206-219.

[CV-86e] R. Cole and U. Vishkin, "Approximate and exact parallel scheduling with applications to list, tree and graph problems", *Proc. Twenty Seventh Annual Symp. on Foundations of Computer Science*, 1986, 478-491.

[CV-86f] R. Cole and U. Vishkin, "Faster optimal parallel prefix sums and list ranking", TR 56/86, Dept. of Computer Science, Tel Aviv Univ., 1986; also, TR 277, Dept. of Computer Science, Courant Institute, NYU, 1987.

[G-86] H. Gazit, "An optimal randomized parallel algorithm for finding connected components in a graph", *Proc. Twenty Seventh Annual Symp. on Foundations of Computer Science*, 1986, 492-501.

[HCS-79] D.S. Hirschberg, A.K. Chandra and D.V. Sarwate, "Computing connected components on parallel computers", *Comm. ACM* 22 (1979), 461-464.

[KRS-85] C.P. Kruskal, L. Rudolph and M. Snir, " Efficient parallel algorithms for some graph problems", *Proc. Int. Conf. on Parallel Processing*, 1985, 180-185.

[Lo-85] L. Lovasz, "Computing ears and branchings in parallel", *Proc. Twenty Sixth Annual Symp. on Foundations of Computer Science*, 1985, 464-467.

[MR-85] G. Miller and J. Reif, "Parallel tree contraction and its application", *Proc. Twenty Sixth Annual Symp. on Foundations of Computer Science*, 478-489.

[MSV-86] Y. Maon, B. Schieber and U. Vishkin, "Parallel ear decomposition search (EDS) and *st*-numbering in graphs", *Theoretical Computer Science*, to appear.

[SJ-81] C. Savage and J. Ja'Ja', "Fast, efficient parallel algorithms for some graph problems", *SIAM J. Computing* 10 (1981), 682-691.

[SV-82] Y. Shiloach and U. Vishkin, "An $O(\log n)$ parallel connectivity algorithm", *J. Algorithms* 3 (1982), 57-67.

[ScV-87] B. Schieber and U. Vishkin, "On finding lowest common ancestors: simplification and parallelization", TR 63/87, Dept. of Computer Science, Tel Aviv Univ., 1987.

[TC-84] Y.H. Tsin and F.Y. Chin, "Efficient parallel algorithms for a class of graph theoretic problems", *SIAM J. of Computing* 13 (1984), 580-599.

[TV-85] R.E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm", *SIAM J. of Computing* 14 (1985), 862-874.

[Vi-83] U. Vishkin, "Synchronous parallel computation - a survey", TR 71, Dept. of Computer science, Courant Institute, NYU, 1983.

[Vi-84] U. Vishkin, "An optimal parallel connectivity algorithm", *Discrete Applied Math.* 9 (1984), 197-207.

[Vi-85] U. Vishkin, "On efficient parallel strong orientation", *Information Processing Letters* 20 (1985), 235-240.

[W-79] J.C. Wyllie, "The complexity of parallel computation," TR 79-387, Department of Computer Science, Cornell University, Ithaca, New York, 1979.

Appendix: the vertex table

The vertices are stored in a table T. For each vertex v, $T(v)$ is the name of the supervertex currently containing v. We show how to maintain T, using a total of $O(n\,\alpha(m,n) + m)$ operations and $O(\log n)$ time, so that it takes time $O(\alpha(m,n))$ using a single processor to determine $T(v)$. Henceforth, we denote $\alpha(m,n)$ by α. The vertices undergo alternating sets of UNION and FIND operations. Each set of operations is performed in parallel. Each set of UNION operations is provided in the following form. The names of the new supervertices are provided in an array; also each old supervertex has the name of the new supervertex. In addition, it is guaranteed that every set of UNIONs reduces the number of supervertices by a factor of at least 2. This makes the problem considered here only an instance of the general UNION-FIND problem (see, e.g., [AHU-74]).

We maintain the vertex/supervertex table as a forest of height exactly $\alpha + 1$. The leaves represent vertices. The roots, level 1 nodes, represent the current supervertices. Internal nodes represent supervertices created at some of the earlier phases of the processing. Initially, each tree is a chain of $\alpha + 1$ copies of the same vertex.

After every phase, the forest is updated as follows. We introduce new roots, corresponding to the supervertices created in this phase. Consider the new root corresponding to the new supervertex v; its children are the following 'old' roots: those old roots corresponding to the supervertices that were combined to form supervertex v. We update the parent pointers for the level 2 nodes by shortcutting over the level 1 nodes; the new roots form the new level 1 nodes. This process takes time $O(1)$ and the number of operations performed is proportional to the size of level 2. On occasion, we update the level $i > 1$ nodes (and consequently the parent pointers for the level $i + 1$ nodes) as follows. We create a new instance of level $i-1$: it is a copy of the old level $i-1$. The new level i is the old level $i-1$ with the parent pointers changed: the parent of a new level i node (an old level $i-1$ node) is the corresponding node at level $i-1$. We then update the parent pointers of the nodes at level $i+1$ by shortcutting over the old level i. The number of operations performed by this process is proportional to the size of level $i+1$; it takes time $O(1)$.

In order to easily perform the update of a level it is convenient to store the nodes at each level in an array (we can then perform the processor allocation for the update in $O(1)$ time). In order to do this, it suffices, after each phase, when new roots are introduced, to place these new roots in an array. But these roots are simply the new supervertices. And this is how the new supervertices are provided by Step 3.

We observe that the sizes of the levels, going down the tree (towards the leaves), are non-decreasing. A good intuitive guide is to view the sizes as being rapidly increasing (although this is not strictly correct all the time). Thus we can afford to update successively deeper levels of the tree less and less frequently. At this point, it is helpful to recall the definition of Ackerman's function and its inverse.

$A(1, j) = 2^j \quad$ for $j \geq 1$

$A(i, 1) = A(i-1, 2) \quad$ for $i > 1$

$A(i, j) = A(i-1, A(i, j-1)) \quad$ for $i, j > 1$.

We define $A(i, 0) = 1$, for all i. (This last definition is merely for convenience. It is used in Lemmas 1 and 2, below.)

We define $\alpha(m, n)$ to be the least i such that $A(i, \lceil m/n \rceil) \geq n$.

Since the number of supervertices decreases by a factor of 2 following each phase, the size of level 1 is bounded, in turn, by n, $n/2$, $n/2^2$, $n/2^3$,... When the size of level 1 is first bounded by $n/(2^2)^2$ we update level 2 (reducing the size of level 2 to at most $n/(2^2)^2$); again, when it is first bounded by $n/(2^{2^2})^2$ we update level 2 (reducing the size of level 2 to at most $n/(2^{2^2})^2$), and so on (the outermost squaring is present for technical reasons; for intuitive understanding it can be ignored). More generally,

level i, for $1 < i \leq \alpha$, is updated when the size of the $i-1$st level is first bounded by $n/[A(i, r)]^2$, for $r = 1, 2,...,$ in turn.

Lemma 1: For $i > 1$, between two updates of level $i+1$ (which reduce level $i+1$ from size at most $n/[A(i+1, r)]^2$ to size at most $n/[A(i+1, r+1)]^2$, for some $r \geq 0$), level i is updated at most $A(i+1, r)$ times.

Proof: Between these two updates of level $i+1$, level i is reduced from size at most $s_1 = n/[A(i, A(i+1, r-1))]^2$ (for $r > 0$), and from size $n = n/[A(i, 0)]^2$ (for $r = 0$), to size at most $s_2 = n/[A(i, A(i+1, r))]^2$. An update of level i reduces it from size at most $n/[A(i, j)]^2$ to size at most $n/[A(i, j+1)]^2$; thus reducing level i from size at most s_1 to size at most s_2 requires at most $A(i+1, r) - A(i+1, r-1)$ updates of level i if $r > 0$, and at most $A(i+1, r)$ updates of level i if $r = 0$. The lemma follows. $\square$

Lemma 2: Between two updates of level 2 (which reduce level 2 from size at most $n/[A(2, r)]^2$ to size at most $n/[A(2, r+1)]^2$, for some $r \geq 0$) there are at most $2A(2, r)$ updates of level 1 for $r>0$, and at most 4 updates of level 1 for $r=0$.

Proof: Each update of level 1 reduces its size by a factor of at least 2. Thus, between these two updates of level 2 there are at most $2\log [A(2, r+1)]$ updates of level 1. But, for $r > 0$,

$\log [A(2, r+1)] = \log [A(1, A(2, r))] = A(2, r)$, and for $r = 0$, $\log [A(2, r+1)] = 2$. The lemma follows. □

Lemma 3: Level α is updated less than m/n times.

Proof: Following $\lceil m/n \rceil$ updates level α would have size at most $\frac{n}{[A(\alpha, \lceil m/n \rceil)]^2} < 1$. The lemma follows. □

Lemma 4: The total number of operations used to perform all the updates is $O(n\alpha + m)$.

Proof: Each update to level α uses $O(n)$ operations; so, by Lemma 3, the updates to level α use $O(m)$ operations altogether. By Lemma 1, the updates to level i, $1 \le i < \alpha$, for each r use $O\left(\frac{n}{A(i+1,r)^2} A(i+1,r)\right)$ operations and in total, $O\left(\sum_{r \ge 0} \frac{n}{A(i+1, r)}\right)$ operations. But $\sum_{i>0} \sum_{r>0} \frac{n}{A(i+1, r)} = O(n)$ and $\sum_{i>0} \frac{n}{A(i+1, 0)} = O(n\alpha)$. □

Each update takes $O(1)$ time and there are $O(\log n)$ updates to all the levels (for the number of updates at level 1 is $O(\log n)$, at level 2 is $O(\log^* n)$, and in general, decreases by a factor of at least 2 from level to level; in fact it decreases by much larger factors). To know when to perform an update, we need to maintain a count of the number of nodes at each level of the forest. Level i acquires the size of level $i-1$ whenever level i is updated. Also recall that the nodes at each level are stored in an array; whenever the size of level i is reduced, the new level i nodes must be compressed into contiguous locations in the array. But this is easy to do, for a new level i is merely a copy of level $i-1$.

Thus, over the whole algorithm, maintaining the vertex table requires $O(n\alpha + m)$ operations and $O(\log n)$ time. The table uses space $O(n\alpha)$ space. Clearly, each FIND takes $O(\alpha(m, n))$ time.

Remark. We can reduce the operation count for maintaining the vertex table to $O(n + m)$ operations. It suffices to keep just one instance of 'identical' levels, that is levels of the same size. We leave it to the interested reader to verify the claimed complexity bound.

NYU COMPSCI TR-291 c.2
Cole, Richard
Approximate parallel
scheduling. Part II

NYU COMPSCI TR-291 c.2
Cole, Richard
Approximate parallel
scheduling. Part II

